Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1723 – 1732

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Literate program execution for teaching computational science

Sébastien Li-Thiao-Té[a,*]

[a] *Université Paris 13, CNRS, UMR 7539 LAGA*
99, avenue Jean-Baptiste Clément, F-93 430 Villetaneuse, France

Abstract

Class material for computer science courses often contains algorithms and code snippets, as well as the results of their execution. Usually, these are written and tested outside the source document then included via copy-and-paste. Making sure that the code compiles and that the results really correspond to the included code is the teacher's responsibility.

Using techniques and ideas from literate programming, we propose to include source code and executable instructions inside the source document. To support this, we have implemented Lepton which is a tool for extracting source code, compiling, executing, and including the results of the documented programs. Consequently, copy-and-paste is eliminated and code output is guaranteed to be up-to-date with source code. This manuscript was written with Lepton.

Keywords: Tools to aid in teaching, reproducible research, literate programming, executable papers

1. Introduction

In many areas of Computer Science and Applied Mathematics, algorithms are not only designed but also implemented. Consequently class material usually contains both the abstract representation of the algorithm and an implementation example. For programming courses in particular, it is important that the provided examples be correct and compliant with their description or specification.

Correctness of code snippets encompasses several notions. First, the source code described in the documentation should coincide with the code executed by the computer. Then this code should be syntactically correct, i.e. it should compile. Finally, the code should yield the correct results. These properties are similar to the requirements of an "executable paper" for "reproducible research", so a similar solution can be proposed for writing class material: write the code examples directly inside a LaTeX file, and have a robot extract the code, compile, run and include the results back to the LaTeX file. Lepton[1] is such a robot.

The aim of this manuscript is to demonstrate how to use Lepton in Computer Science and Statistics courses. We provide the Lepton manual and a tutorial in Section 2. We then discuss how to use Lepton to write and document source code in Section 3 and material for Statistics courses in Section 4. A comparison with existing software is provided in Section 5.

*Corresponding author
Email address: lithiao@math.univ-paris13.fr (Sébastien Li-Thiao-Té)

doi:10.1016/j.procs.2012.04.190

2. The Lepton manual

2.1. Tutorial

Lepton processes files written in LaTeX-like syntax. To write a "hello world" manuscript, the first step is to write a `hello.nw` file containing:

Code chunk 1: `<<hello.nw>>`

```
\documentclass[paper=a7]{scrartcl}
\usepackage[width=7cm,height=10cm]{geometry}
\input{lepton.sty}
\begin{document}
The code below sends "hello world" instructions to the \verb ocaml  interpreter.
<<hello_world -exec ocaml>>=
let msg = "Hello world.";;
print_string(msg); print_newline();;
@
\end{document}
```

The second step is to apply Lepton. This tool splits the file into documentation and source code, executes instructions where specified, and embeds the results. Lepton turns `hello.nw` into a legitimate LaTeX document `hello.tex`. When processing a file, Lepton outputs the name of each encountered code snippet and how it deals with it.

Code chunk 2: `<<hello.tex>>`

```
lepton.bin hello.nw

hello_world (part 1): chunk as ocaml, exec with ocaml, output as text,
```

The last step is to compile using `pdflatex`. The `-shell-escape` option enables colorful pretty-printing with the `minted` LaTeX package. The resulting PDF file is displayed in Figure 1.

Code chunk 3: `<<hello.pdf>>`

```
pdflatex -interaction batchmode -shell-escape hello.tex

This is pdfTeX, Version 3.1415926-1.40.10 (TeX Live 2009/Debian)
 \write18 enabled.
entering extended mode
/usr/bin/pygmentize
```

2.2. Usage and command-line options

```
lepton [-o texname] [-env envname] [filename]
```

Lepton uses `filename` as the input file name, or standard input when absent.

-o texname sets the name of the output LaTeX file.

-env envname uses `envname` instead of the default `minted` environment. See Section 2.5 for details.

2.3. Syntax

The syntax used in Lepton is inspired by the syntax of Noweb files [2] because of its simplicity. Lepton files are plain-text LaTeX files which may contain special blocks called *code chunks*. In Lepton, code chunks start with a chunk header of the form `<<header>>=` at the beginning of the line, and end with `@` at the beginning of the line. The chunk header is parsed as a blank separated command line. The first word is the chunk name. The following words are interpreted as chunk options. These control the output and interpretation of the chunk contents. Code chunks can appear in any order.

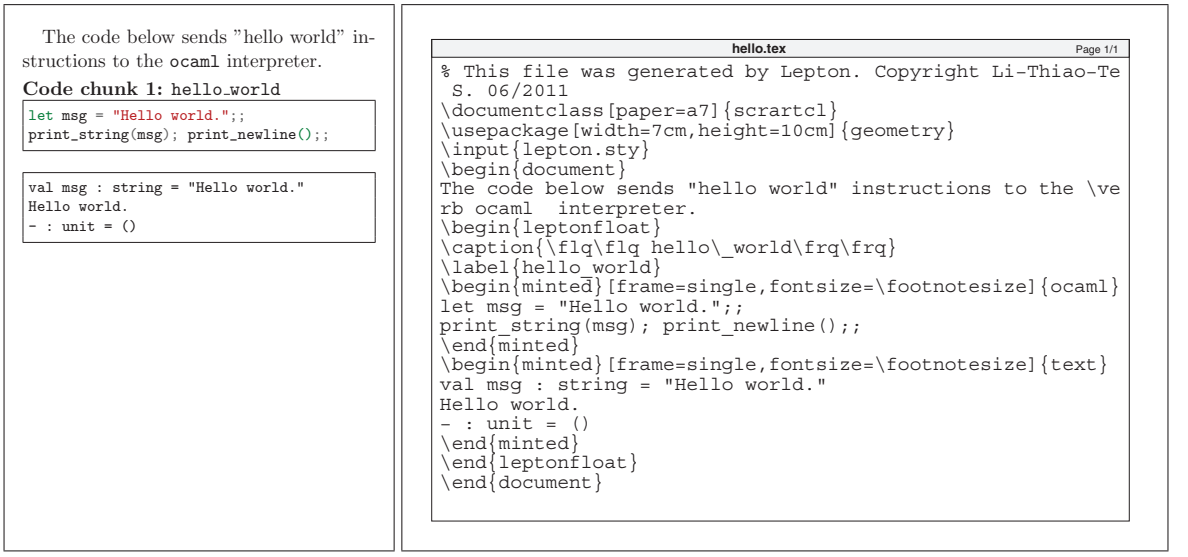

Figure 1: PDF file (left) and LaTeX source (right, rendered by a2ps) produced from hello.nw by Lepton.

Code chunks can contain references that are written as `<<chunkname>>`. The chunk reference is replaced by the concatenation of all chunks with the same name. The amount of whitespace before the chunk reference is used to set the indentation level: it is prepended to all lines when expanding the reference.

Code chunks can contain other code chunks. This allows embedding of Lepton files inside other Lepton files, such as the `hello.nw` example in Section 2.1.

Two directives in LaTeX syntax are interpreted by Lepton. We define a `\Lexpr{interpreter}{code}` macro for direct inclusion of results in the LaTeX documentation. We also define a `\Linput{filename}` directive for including Lepton files and interpreting their contents. The Lepton manual is included in this document with `\Linput`.

2.4. Interpretation of code snippets

The contents of code chunks are interpreted as specified by the options in the chunk header:

- `-write -nowrite` : write the chunk contents to disk and use the chunk name as file name. Default: `-nowrite`,
- `-expand -noexpand` : expand chunk references in the documentation. Default: `-noexpand`,
- `-exec interpreter` : execute the chunk contents in an external interpreter. Default: `none`, i.e. do not execute,
- `-chunk format -output format` : indicate the format of the chunk contents and the chunk output for pretty-printing. By default, the format is verbatim text. Special values are `verbatim` (the output is formatted LaTeX code intended for direct inclusion) and `hide` (the output is not included in the produced `tex` file).

Lepton interprets the source file sequentially. For each chunk, the references are recursively expanded, then the chunk contents are optionally written to disk, and the chunk contents are optionally sent to the external interpreter. In particular, written files and definitions sent to an interpreter are available for the subsequent code chunks.

The `interpreter` specified with `-exec` or `\Linput` is a session or process name. If it corresponds to a process already open by Lepton, the process will be reused. Otherwise, the interpreter name is matched (by prefix) to a list of known intepreters and a new instance is launched. Lepton currently supports the UNIX shell, OCaml, Python, R and there is preliminary support for Matlab. Several sessions of the same process can be open concurrently, e.g. `shell1, shell2, shellbis`.

Other programming languages, notably compiled languages such as C/C++, can be used in Lepton by writing the source code to disk and using the `shell` interpreter to compile and execute the programs. To use a makefile, put the text into a chunk, write the chunk to disk and execute with `shell`.

Options that are set for a code chunk are propagated to the following chunks of the same name. `lepton_options` is a reserved chunk name for setting default options. For example, `<<lepton_options -write -chunk ocaml>>=`

sets the default behavior to writing all chunk contents to disk, and formatting the chunk contents as OCaml code. The chunk contents are ignored.

2.5. LaTeX format and pretty-printing

Lepton relies on LaTeX for formatting the documentation. Lepton wraps the chunk contents and its output in a LaTeX environment called `leptonfloat`, which is based on the `float` package (see Figure 1). Consequently,

- a caption is automatically included based on the chunk name,
- labels and indexes are automatically defined, the `hyperref` package can be used to link to chunk definitions,
- for each chunk reference, Lepton automatically adds a hyperlink to the corresponding chunk definition.

A list of all code chunks can be generated with `\lelistoflistings` and an index of code chunks with `makeidx`.

The chunk contents and the chunk output are independently formatted according to their respective options. For pretty-printing, we use the `minted` package in combination with the Python `Pygments` beautifier [3] to provide colorful syntax highlighting for many languages (See the rendered `hello.pdf` in Section 2.1). When not available, the `minted` environment can be replaced with another LaTeX environment via a command-line option to Lepton.

The current version of Lepton includes preliminary HTML output support.

2.6. Current implementation and availability

The current implementation is written as a Lepton file with source code in the OCaml programming language. The Lepton code can be compiled to native code for speed on many architectures, and requires no external libraries.

Standalone binaries are available for GNU/Linux 32-bit and 64-bit platforms and can be downloaded from `http://www.math.univ-paris13.fr/~lithiao/Lepton.html`. For other platforms such as Windows, the mechanism for external command execution has not been ported yet.

3. Writing and documenting code

3.1. Embedding source code in the documentation

Lepton follows the literate programming paradigm; Lepton files are documents with embedded source code rather that source code with embedded comments. Code and its documentation are contained in a single file and preferably in the same location in that file. By using chunk references, source code can be divided into chunks that are meaningful and easy to explain. When writing course material, the teacher can include the source code next to the corresponding formal description of the algorithm. For instance, let us describe the merge sort algorithm.

The merge sort algorithm is a divide and conquer algorithm for sorting lists (function `merge`, see implementation). Conceptually, it operates with the following steps:

- when the list contains 0 or 1 elements, then it is already sorted,
- when the list contains more than 2 elements, we can divide it in two (function `split`, see implementation),
- the merge sort algorithm is applied on the two sub-lists,
- combining two sorted lists can be implemented efficiently (function `fusion`, see implementation).

In the following, we use the OCaml toplevel interpreter [4]. Lepton opens a single process called `caml`, sends the chunk contents to this process, retrieves the output of the toplevel interpreter and embeds them in the documentation. In particular, the OCaml interpreter outputs the inferred type of the functions.

Lists in OCaml are defined as either the empty list `[]` or a construct of the form `h :: t` where `h` is a single element (the list head) and `t` is a list (the list tail). Such constructs can be used in pattern-matching to select a code path based on the structure of the argument.

The `split` function operates recursively on lists:

- when a list contains two elements followed by a tail (`a :: b :: c`), we split the tail into two sub-lists `l1` and `l2` and return the lists `(a :: l1)` and `(b :: l2)`
- when a list contains 0 or 1 elements, the result is trivial. Note that this case only occurs when splitting a tail `c`. The `merge` function calls the `split` function with at least 2 elements.

Code chunk 4: <<caml>>
```
let rec split = function
  | a :: b :: c -> let l1,l2 = split c in (a :: l1), (b :: l2)
  | a :: [] -> [a],[]
  | [] -> [],[]
;;

val split : 'a list -> 'a list * 'a list = <fun>
```

The type inferred by the OCaml toplevel means that the `split` function takes a list of type `'a list` and returns a pair of lists of type `'a list * 'a list`. Functions in OCaml are polymorphic; `'a` is a type variable. This function prototype means that the input and outputs of the `split` function must contain elements of the same type.

To combine two sorted lists into a sorted list, the `fusion` function compares the two list heads `a` and `b`, selects the smaller element, and recursively combines the remaining elements. When one list is empty, the operation is trivial.

Code chunk 5: <<caml>>(part 2)
```
let rec fusion = function
  | a :: ra, b :: rb when a < b -> a :: fusion (ra, b::rb)
  | a :: ra, b :: rb (* when a >= b *) -> b :: fusion (a::ra, rb)
  | [],l | l,[] -> l
;;

val fusion : 'a list * 'a list -> 'a list = <fun>
```

To sort a list, the `merge` function calls `split`, sorts the results with a recursive call, then applies `fusion` to the sorted sub-lists. Its operation on the empty list or on a list containing only one element is trivial; this ensures that the algorithm terminates.

Code chunk 6: <<caml>>(part 3)
```
let rec merge = function
  | [] -> []
  | a :: [] -> a :: []
  | l -> let l1,l2 = split l in fusion (merge l1, merge l2)
;;

val merge : 'a list -> 'a list = <fun>
```

When using the OCaml interpreter, function definitions are evaluated and Lepton automatically embeds the inferred types. This procedure guarantees that the *displayed* code is syntactically correct, and we can check visually that the code has the correct type. Moreover, we can also immediately check that the code is correct on an example:

Code chunk 7: <<caml>>(part 4)
```
let l = [3;5;10;9;4;2;6;7;8;1];;
let result = merge l;;

val l : int list = [3; 5; 10; 9; 4; 2; 6; 7; 8; 1]
val result : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

3.2. Executing compiled code

Lepton can be used with compiled programs by first writing the source code to disk, then executing the compilation instructions with the UNIX shell. In this section, several aspects of C programming using Lepton are illustrated with an example inspired by the GNU scientific library [5]. In addition to source code, Lepton files document the compilation instructions and provides the same coherence and correctness guarantees.

C is a compiled language, so programs must be written to disk, and compiled to binary format before execution. The code for the example is defined below — inside the Lepton file — with chunk option `-write` which instructs Lepton to write the chunk contents to the disk.

Code chunk 8: <<example.c>>
```
includes

<<includes>>
int
main (void)
{
    double x = 5.0;
    double y = gsl_sf_bessel_J0 (x);
    printf ("J0(%g) = %.18e\n", x, y);
    return 0;
}
```

The code is compiled by gcc using the following instruction:

Code chunk 9: <<shell>>
```
gcc -Wall -I/usr/local/include -lgsl -lgslcblas -lm example.c -o a.out 2>&1
```

Note that gcc compiler writes errors to stderror which needs to be redirected to stdout for inclusion. The output is empty when no error occurs. In this case, we ensure that the code in example.c is syntactically correct.

The example program computes the value of the Bessel function $J_0(5)$:

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}$$

and produces the following output.

Code chunk 10: <<shell>>(part 2)
```
./a.out

J0(5) = -1.77596771314338292e-01
```

In the code chunk example.c, we have extracted the include directives to provide more readability. The same directives can be reused in other programs. This mechanism can also be used to include a common piece of text such as a license or copyright at the beginning of all the source files.

Code chunk 11: <<includes>>
```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
```

When dealing with larger projects, we can use makefiles to build executables more easily. The makefile can be defined at the beginning of the Lepton file.

Code chunk 12: <<makefile>>
```
build_targets

CC=gcc -Wall -I/usr/local/include -lgsl -lgslcblas -lm
<<build_targets>>
clean:
rm *.c *.o || TRUE
rm a.out example.out
```

For each build target, we define the instructions in the <<build_targets>> chunk.

Code chunk 13: <<build_targets>>
```
example:
$(CC) example.c -o example.out
```

When processing the documentation, Lepton first assembles all the parts in the `build_targets` chunk then writes the makefile to disk. Chunks defined after the makefile can assume that the file is written, and consequently build targets with `make`. Note that the source files can be removed by `make clean` because they are defined in the Lepton file.

Code chunk 14: `<<shell>>(part 3)`

```
make example
./example.out

gcc -Wall -I/usr/local/include -lgsl -lgslcblas -lm example.c -o example.out
J0(5) = -1.775967713143382920e-01
```

4. Writing course material and exercises

4.1. Generating examples and exercises

Lepton provides access to the full capabilities of any programming language. In particular, we can use a random number generator to produce random exercises. The following example comes from an undergraduate level statistics course given at Université Paris 13 in the school year 2011-2012 and uses R for statistics computations. We generate a dataset with the following code.

Code chunk 15: `<<r>>`

```
data0 = floor(runif(20)*100)/100 # generate 20 uniformly distributed numbers
cat(data0,sep=", ")
```

0.85, 0.81, 0.75, 0.75, 0.21, 0.58, 0.4, 0.5, 0.08, 0.88, 0.12, 0.72, 0.82, 0.98, 0.82, 0.15, 0.7, 0.29, 0.93, 0.17

When writing the questions, variables defined in the chunk r can be used in LaTeX. For example, the third number in the dataset can be inserted with `\Lexpr{R}{cat(data0[3])}`. This produces 0.75.

To generate random questions, we define a list of four quantities that we want the students to compute, and select two of them randomly out of the four possibilities.

Code chunk 16: `<<r>>(part 2)`

```
functions_all = c("mean","variance","standard deviation","median")
functions_sel = sample(functions_all,2)
```

Questions can be written as usual in LaTeX, for instance with the `itemize` environment[1]:

- Compute the median of the dataset. (`\Lexpr{R}{cat(functions_sel[1])}`)
- Compute the mean of the dataset. (`\Lexpr{R}{cat(functions_sel[2])}`)

Randomized exercises emphasize the generic nature of mathematics. They force the students to not expect special cases: datasets do not contain integers in general, the mean of the dataset is not an integer, numbers are not always positive, etc. When teaching statistics, random exercises are a way to build experience for the students as they provide examples with many different shapes.

Generating random exercises has many advantages when dealing with copying during exams. Obviously, the true answers are different, but also the questions cannot be predicted in advance. If the subjects are numbered, it is easy to correlate the subject number with the location in the examination room, while still preserving anonymity. It is still possible for one student to reproduce the reasoning of his/her neighbour. However, successful adaptation of the method requires a deep level of understanding of the question and its solution; which is worth some points in our opinion. Randomizing the order of the exercises when these are unrelated encourages students to start with what they are comfortable with, and not with the order presented.

Lepton is currently in place at Université Paris 13 for generating a different subject for each student in end-of-semester exams as well as exercise sheets for statistics labs. In the current system, numerical values, names and the order of exercises can be randomized.

[1] Obviously, chunk contents are hidden when printing an exam subject.

4.2. Generating solutions and figures

Generating random exam subjects puts a strain on the teacher when grading. Fortunately, the same capabilities of the programming language can be used to generate the solutions to the exercises. By setting the default options, code corresponding to solutions can be easily hidden throughout the document For instance, the following code computes all the necessary quantities:

Code chunk 17: `<<r>>(part 3)`

```
cat("mean = (", data0[1],"+",data0[2],"+ ... ) /",length(data0),"=", mean(data0))
cat("\nvariance", var(data0))
cat("\nstandard deviation", sd(data0))
cat("\nmedian", median(data0))
```

```
mean = ( 0.85 + 0.81 + ... ) / 20 = 0.5755
variance 0.09364711
standard deviation 0.3060181
median 0.71
```

Generating full solutions with intermediate results can be tedious. However, it helps to spot mistakes when grading and full solutions can be printed for the students. We suggest that solution sheets be used for grading instead of the student's paper. Solution sheets can be designed with a standard and concise format with wide margins for grades.

Teachers can also programmatically generate figures to include in course material, slides, or solutions. In Figure 2, we illustrate the variability of histograms with respect to the choice of the breakpoints. We generate several histograms with random breakpoints from the dataset generated in code chunk r. The (hidden) code chunk directly includes the generated figures in the LaTeX document.

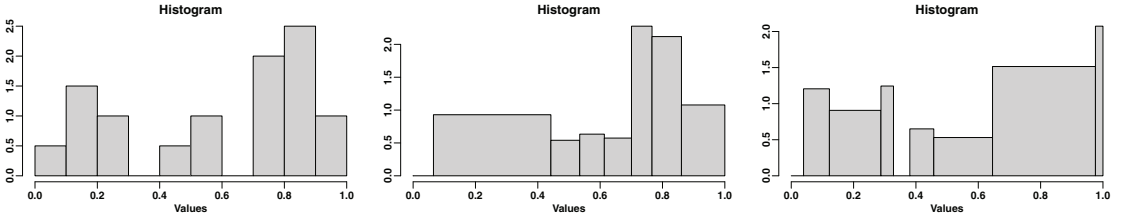

Figure 2: Three histograms programmatically generated from the dataset defined in code chunk 15.

4.3. Reviewing student assignments

Students (and teachers) can also benefit from Lepton for writing their assignments and technical reports. Students gain all of Lepton's features for writing well-documented reports: automatic inclusion and formatting of the source code, automatic inclusion of results.

When reviewing a student paper, the teacher automatically receives the source code and instructions for compiling and using the code. Moreover, by applying Lepton in a blank environment, the teacher can easily check that the code runs as expected.

5. Comparison with existing software

5.1. Software for e-learning

Most e-learning software can produce some level of random exercises. For example, there are software solutions for producing multiple choice questionnaires that can even interface with a scanner for automatic grading. For exercises in mathematics, the WIMS [6] and MOODLE [7] enable the teacher to write online exercises with their answers in a restricted programming language (based on Javascript for WIMS, simple formulae for MOODLE). The software

can automatically handle randomization, grading, and also distribute course material to students, track their progress, etc.

Lepton is similar to Adessowiki [8] which offers a collaborative editing system or Wiki in ReST syntax, Python code snippets and automatic inclusion of images produced by the function `mmshow`. Compared to WIMS and MOODLE, fewer tasks are automated and the framework offers limited interactivity. However, Lepton is a tool to circumvent the limitations of existing frameworks, such as a specific programming language with restricted features, or a specific documentation syntax such as ReST or HTML. As such, it can be better suited for assignments such as a dissertation or a computer program rather than simpler exercises where other available software provide sufficient features.

5.2. Literate programming

The concept of "literate programming" was invented by D. E. Knuth with the WEB program [9], which he used to implement TEXand METAFONT. Most of the successors of WEB such as FunnelWEB [10], NuWEB [11] and Noweb [2] still follow its design principles. The tool is divided in two main operations: `tangling` which extracts source code and `weaving` which produces documentation. In particular, the syntax used in Lepton is derived from the syntax in Noweb files.

Sweave [12] is a Noweb-like implementation of literate programming targeted to the R environment for statistical computing, and especially suited for producing reports of statistical analyses. Sweave already provides most of the features in Lepton, but is restricted to the R system.

Lepton's features are a mix of of these literate programming tools. We took the syntax from Noweb, but chose a one-step procedure similar to NuWEB and Sweave, instead of the tangling/weaving approach. As in Sweave, Lepton documents can be used to emulate a terminal session and communicate with an external interpreter. However, Lepton can be used with interactive interpreters such as OCaml, Python and Matlab. In particular, the UNIX shell interpreter makes it possible to use Lepton with virtually any programming language or software.

Literate programming has already been proposed for teaching computer science in [8, 13] for example. Such pedagogical initiatives can be reproduced with other literate programming tools and frameworks — including Lepton— because the focus is on problem solving rather than programming tools. Nevertheless, we believe that Lepton offers larger possibilites by enabling the communication with external interpreters in general. This enables the documentation of instructions for compiling, testing and using programs in the literate programming paradigm. It also exposes the parameters and instructions used for generating output, i.e. the provenance information necessary for easily be reviewing and reproducing the contents of the report.

6. Conclusions

Lepton is a tool for documenting source code in the literate programming paradigm. Consequently, it is well-suited for writing course material in computer science. Additionally, access to external interpreters and in particular the UNIX shell makes it possible to systematically compile and test source code in Lepton files and provides strong guarantees on its correctness and coherence.

Lepton's features make it possible to programmatically generate PDF documents. This can be used in course material for generating solutions to exercises or figures. We have also used this for generating random exercises in exams to limit fraud, for generating large number of exercises for revisions, and for programmatically generating the solution to exercises.

We encourage students to use Lepton for their assignments, so that documented source code and execution instructions are automatically available to the teacher for reviewing. Similarly, we encourage researchers to use Lepton as a medium for writing reproducible research and executable papers.

References

[1] S. Li-Thiao-Té, Lepton User Manual.
URL `http://www.math.univ-paris13.fr/~lithiao/ResearchLepton/Lepton.html`
[2] N. Ramsey, Literate programming simplified, Software, IEEE 11 (5) (1994) 97–105.

[3] G. Brandl, T. Hatch, A. Ronacher, Pygments.
URL http://pygments.org/
[4] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon, The OCaml system (release 3.12): Documentation and user's manual, Institut National de Recherche en Informatique et en Automatique (Jul. 2011).
URL http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf
[5] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, F. Rossi, GNU Scientific Library Reference Manual (3rd Ed.), Network Theory Ltd, 2009.
URL http://www.gnu.org/software/gsl/
[6] X. Gang, Wims: An interactive mathematics server, Journal of Online Mathematics and its Applications.
[7] Moodle.
URL http://moodle.org/
[8] R. Machado, L. Rittner, R. Lotufo, Adessowiki-collaborative platform for writing executable papers, Procedia Computer Science 4 (2011) 759–767.
URL http://www.adessowiki.org
[9] D. Knuth, S. U. C. S. Dept, Literate programming, Center for the Study of Language and Information, 1992.
[10] R. Williams, et al., FunnelWeb user's manual (1992).
URL http://www.ross.net/funnelweb/reference/index.html
[11] P. Briggs, J. D. Ramsdell, M. W. Mengel, S. Wright, K. Harwood, Nuweb Version 1.57 A Simple Literate Programming Tool.
[12] F. Leisch, Sweave: Dynamic generation of statistical reports using literate data analysis, in: W. Härdle, B. Rönz (Eds.), Compstat 2002 — Proceedings in Computational Statistics, Physica Verlag, Heidelberg, 2002, pp. 575–580, iSBN 3-7908-1517-9.
URL http://www.stat.uni-muenchen.de/~leisch/Sweave
[13] B. Childs, Thirty years of literate programming and more? introduction to, experiences with, state of literate programming, TUGboat-TeX Users Group 31 (2) (2010) 183.